Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2015,6
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Regression Verification for Programmable Logic Controller Software

Bernhard Beckert, Mattias Ulbrich,
Birgit Vogel-Heuser, Alexander Weigl

2015

Fakultät für **Informatik**

Regression Verification for Programmable Logic Controller Software

Bernhard Beckert[1], Mattias Ulbrich[1],
Birgit Vogel-Heuser[2], and Alexander Weigl[1]

[1] Karlsruhe Institute of Technology, Germany
beckert@kit.edu, ulbrich@kit.edu, alexander.weigl@student.kit.edu
[2] Technische Universität München, Germany
vogel-heuser@ais.mw.tum.de

Abstract. Automated production systems are usually driven by Programmable Logic Controllers (PLCs). These systems are long-living – yet have to adapt to changing requirements over time. This paper presents a novel method for regression verification of PLC code, which allows one to prove that a new revision of the plant's software does not break existing intended behavior.
Our main contribution is the design, implementation, and evaluation of a regression verification method for PLC code. We also clarify and define the notion of program equivalence for reactive PLC code. Core elements of our method are a translation of PLC code into the SMV input language for model checkers, the adaptation of the coupling invariants concept to reactive systems, and the implementation of a toolchain using a model checker supporting invariant generation.
We have successfully evaluated our approach using the Pick-and-Place Unit benchmark case study.

Keywords: regression verification, symbolic model checking, automated production systems, programmable logic controllers (PLC)

1 Introduction

Motivation and Topic. Automated production systems [34], such as industrial plants and assembly lines, are usually driven by *Programmable Logic Controllers* (PLCs). These computing devices are specially tailored to controlling automated production systems in safety-critical realtime environments. A malfunction may cause severe damage to the system itself or to the payload, or even harm persons within the reach of the system.

The topic of this paper is how to formally verify correctness of the software part of such systems, i.e., the PLC. To be precise, we focus on regression verification of PLC code – as opposed to proving that the PLC code satisfies a functional specification or to proving that the whole production system works correctly. That is, we verify that a version of PLC code after an evolution step shows the same reactive input/output behavior as the old one – allowing only desired deviations that are formally specified. The aim of regression verification is to formally prove that existing (good) behavior is retained during system evolution. The old version serves as specification for the new one.

Our approach and contribution. This work contributes to the field of formal PLC verification by defining a notion of reactive conditional and reactive relational equivalence together with a proof methodology, also in the presence of environment models. Our main contribution is the design, implementation, and evaluation of a regression verification method for PLC code.

A core element of our verification method is a translation of PLC code into the SMV input language for model checkers. Using this translation on both the old and the new software revision, we can construct a formula expressing that intended behavior is retained. We target PLC code written in the two languages Structured Text (ST) and Sequential Function Chart (SFC), which are part of IEC 61131, the industry standard for PLC software [19]; an adaptation to other languages is easily possible.

A further core element is the use of a model checker supporting invariant generation. It is an important insight that this allows the automatic generation of *coupling invariants*, which are a useful tool for efficient regression verification. Accordingly, we have adapted the concept of coupling invariants to the world of reactive systems. And we have implemented our approach in a toolchain using the model checker nuXmv [9]. It supports techniques for predicate abstraction and invariant generation by interpolant inspection [7, 24].

As *full* equivalence of PLC code revisions is *not* the goal in many cases, we have defined and implemented extensions where the behavior of the new code revision may deviate under certain specified conditions and in specified ways.

We have successfully evaluated our approach using the Pick-and-Place Unit, a benchmark case study for the evolution of automated production systems with several evolution scenarios [35]. We were able to demonstrate our method's feasibility for practical evolution scenarios and its ability to uncover regression bugs.

PLCs execute their software in cycles with fixed cycle time. Consequently, PLC code can only cause timing problems if its execution time exceeds the cycle time. Otherwise, the code's exact execution time is irrelevant. Thus, we assume that the cycle time constraint is ensured by other techniques, and we do not consider exact execution time in our method.

Advantages of regression verification for PLC code. The main advantage of regression verification is that no functional or behavioral specification is needed (besides the old code version). In addition, regression verification is particularly well suited for the application area of software in automated production systems for the following reasons.

Automated production systems are designed for long deployment phases, often spanning several decades. But the requirements on production systems change over time. New types of products are to be manufactured. Systems are upgraded to increase throughput or to keep up with technological development. Moreover, flaws in the controlling software or the hardware design may have to be fixed. Production systems therefore frequently *evolve* during their lifetime. Thus, methods and means to safely update their hardware and software – including their PLCs – are of great importance. One has to ensure that a revision does not break existing intended behavior.

As opposed to (regression) testing, regression verification provides an equivalence proof for all possible inputs and not just for individual test cases. Also, while regression testing of PLC software requires either a hardware testbed or an executable hardware model, this is not needed for regression verification. It suffices to provide a formal description of how the hardware has changed during the evolution step (if the hardware has changed at all).

PLC systems can grow rather large, making a (non-regression) correctness verification challenging for fully automatic verification and bisimuation checkers.

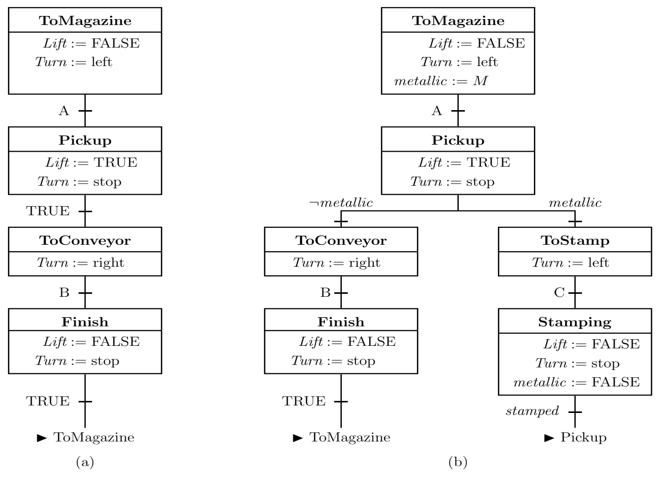

Fig. 1. Sequential Function Charts for the example. (a) Original SFC, (b) SFC after revision.

However, typical changes made during an evolution step are small in comparison to overall system size, so that regression verification is a much easier task.

Structure of this paper. In Section 2, we present a small scenario from our case study as an introductory example. Then, in Section 3, we define the formal framework and introduce notions of equivalence between versions of PLC code. In Section 4, we discuss the use of environment models to avoid false alarms. The core part of our method, i.e., the translation itself and the toolchain are described in Section 5. In Section 6, we present the extensive case study that we used to evaluate our approach. We discuss related work in Section 7 and draw conclusions in Section 8. Some of the ideas presented in the following are adaptations of our regression verification method for imperative programs [11] to the – rather different – world of reactive automated production systems.

2 Introductory Example

As an introductory example, we present a considerably simplified version of a scenario from the case study described in Sect. 6 (see also Fig. 4). A stationary crane moves workpieces from a starting point (A) to one of two target points (B) and (C). In the original version, the plant treats all workpieces in the same way and transports them from the magazine (A) to the conveyor (B).

A new revision of the PLC software is introduced to differentiate the controller's behavior according to the type of workpiece. All metallic workpieces are now first delivered to the stamp (C) where they are treated (signal *stamped*) and are only afterwards delivered to (B). All non-metallic workpieces still go directly to (B). An additional inductive sensor (signal *M*) is installed at (A) to detect whether a workpiece is metallic or not.

Fig. 1 shows sequential function charts (SFCs) for the two versions of the PLC program. The boxes (called *steps*) contain *actions* (blocks of code) and the *transitions* between steps are annotated with guards. In each execution cycle, one step is active and is executed. If at the end of the cycle one of the guards at an outgoing transition is satisfied, the corresponding successor step is made the active step. Otherwise the current step remains active and is repeated in the next cycle. In the example, the actions assign values to output (*Turn*, *Lift*) and internal variables (*metallic*, *stamped*). The guard conditions are Boolean input variables corresponding to sensor input (A, B, C represent input from sensors for crane position) and Boolean internal variables (*metallic*). In the original SFC, the steps correspond to the actions of moving the crane to the magazine, picking up the workpiece, moving the crane to the conveyor, and putting down the workpiece. In the revised SFC, there is a case distinction on *metallic*, and two new steps have been added to move metallic pieces to the stamp and dropping them there. After the workpiece has been stamped, the internal variable *metallic* is set to FALSE, and then the step Pickup becomes active, i.e., from there on the SFC continues in the same way as if a non-metallic workpiece has just been picked up at (A).

Note that this is a simple example. In general, actions and guards can be considerably more complex and contain conditional statements and loops.

In case there are metallic workpieces, the behavior of the PLC is obviously different. But in case that only non-metallic workpieces are ever detected by sensor (M), the new software version should do the same as the old version. So this is a scenario for using regression verification to prove conditional equivalence for the unchanged case.

3 Formalizing Equivalence of PLC Programs

We define a formal framework for the behavior of reactive PLC software together with adequate notions of equivalence between them.

There are various possibilities for defining system boundaries when modeling an automated production system. One can model the whole system or only individual components. Even when focusing on the PLC, one could still include models of peripheral hardware components like connecting data buses. However, our method concentrates on the software that runs on the controller and disregards all effects outside the software for now. Sect. 4 discusses measures to take the environment into consideration.

PLCs are reactive systems with a cyclic data processing behavior, repeating the same control procedure indefinitely. A PLC cycle typically consists of the following steps: (1) read input values, (2) execute task(s), (3) write output values, (4) wait. As reactive systems, PLCs require a notion of equivalence that involves traces, which means that if the old and the new revision are presented with the same *sequence* of input sensor readings, they must produce the same *sequence* of actuator output stimuli.

We call the piece of code that is executed cyclically on the controller a *PLC program*. A PLC program P consists of the instructions Π to be executed and a set of declarations Δ of input, output and state variables. In the introductory example in Sect. 2, the declarations of the program contain the Boolean input (sensor) variables A, B, C and M; *Lift* and *Turn* are output (actuator) variables (the declarations are not shown in Fig. 1).

The internal state of a PLC program consists of an assignment of values to its state variables (in the example, the Boolean variable *metallic*). There is always an implicit state variable *active_step* storing which of the steps in the SFC program is active. The declarations Δ induce an input value space I, an output value space O, and state space S, each as the Cartesian product of the value ranges of the corresponding program variables. In the example, I is $bool \times bool \times bool \times bool$. We assume the initial values of state variables to be determined by their declarations (using default values in case no initial value is given), i.e., the initial state $s_0 \in S$ is fixed by Δ.

Definition 1 (Semantics of PLC programs). *The semantics $\rho(P)$ of a PLC program P is a state transition function $\rho(P) : S \times I \to S \times O$.*

The semantics $\rho(P)$ depends on the instructions in Π. These may read from the state and the input variables (in S and I) and write to the state variables and to the output variables (in S and O).

To be able to consider the effects of a PLC program over time, the above definition needs to be extended to sequences of inputs and outputs. We denote infinite sequences of elements in I (ω-words) with $\bar{i} \in I^\omega$; their components are accessed using subscript indices, i.e., $\bar{i} = \langle i_1, i_2, \ldots\rangle$. The PLC program as a stateful system needs an initial state s_0 from which it is launched. As mentioned above, s_0 is determined by the variable declarations Δ.

Definition 2 (Trace Semantics of PLC Programs). *The behavior $b(P)$ of a PLC program P with initial state $s_0 \in S$ is the function $b(P) : I^\omega \to O^\omega$ defined by $b(P)(\langle i_1, i_2, \ldots\rangle) = \langle o_1, o_2, \ldots\rangle$ where $(s_n, o_n) = \rho(P)\big((s_{n-1}, i_n)\big)$ for all $n \in \mathbb{N}_{\geq 1}$.*

This definition says that starting from the initial state s_0, the PLC program is executed repeatedly, applying in each cycle $\rho(P)$ to its current state s_{n-1} and the input tuple $i_n \in I$ to produce the output tuple $o_n \in O$ and the new state s_n.

Trace semantics use the internal state in the definition, but when taking an outside look at the semantics, it defines input/output behavior and does not make statements about the internal state space. This is relevant for our initial definition of equivalence where it is required that two programs produce identical traces.

Definition 3 (Trace Equivalent PLC Programs). *Two PLC programs P, Q with the same declarations $\Delta_P = \Delta_Q$ are called* perfectly equivalent *if they produce the same output sequence when presented with the same input sequence, i.e., $b(P)(\bar{i}) = b(Q)(\bar{i})$ for all $\bar{i} \in I^\omega$.*

They are called conditionally equivalent *modulo the condition $\varphi : I^\omega \to bool$ if they produce the same result for all input sequences that satisfy condition φ, i.e., if $\varphi(\bar{i})$ then $b(P)(\bar{i}) = b(Q)(\bar{i})$ for all $\bar{i} \in I^\omega$.*

It is intuitively evident that replacing a PLC with a new revision whose program is trace equivalent to the original program does not change the observable behavior of the plant, provided everything else remains unchanged and timing effects are left aside.

Conditional equivalence relaxes the strict notion of perfect equivalence by requiring the same output sequence only if a condition φ holds. Intuitively this means that replacing a PLC with a new revision whose program is conditionally equivalent to the original program modulo φ does not change the plant's behavior at least for those traces where all sensor signal readings satisfy φ.

The example given in Sect. 2 is an example of conditional equivalence: The modified controller software (Fig. 1) is conditionally equivalent to the original version modulo the condition that every encountered workpiece is non-metallic. This condition can be expressed in Linear Temporal Logic (LTL [26]) as $\varphi_{\text{non-metallic}}(\bar{\imath}) = \mathbf{G}\,\neg M$ recalling that M is the signal from the inductive metal detection sensor.

Perfect and conditional equivalence use equality to compare input and output traces. There are many cases, however, where full equality is not required or not appropriate. Equality of outputs may not be required for outputs relating to non-critical components of the system. And equality may not be the appropriate relation if the sensors and/or actuators of the plant have been modified, and thus the input/output spaces of the program revisions are different. It is therefore necessary to generalize the equivalence notion. To this end, we introduce binary relations $\sim_{in}$ and $\sim_{out}$.

Definition 4 (Relational Equivalence of Controllers). *Two PLC programs P, Q with declarations Δ_P resp. Δ_Q are called* relationally equivalent *modulo relations $\sim_{in} \subseteq I_P^\omega \times I_Q^\omega$ and $\sim_{out} \colon O_P^\omega \times O_Q^\omega$ if they produce related output sequences when presented with related input sequences, i.e.,*

$$\textit{if } \bar{\imath} \sim_{in} \bar{\imath}' \textit{ then } b(P)(\bar{\imath}) \sim_{out} b(Q)(\bar{\imath}') \textit{ for all } \bar{\imath} \in I_P^\omega, \bar{\imath}' \in I_Q^\omega.$$

Note that conditional equivalence can be expressed as relational equivalence (if $I_P = I_Q$ and $O_P = O_Q$) by choosing $\bar{\imath} = \bar{\imath}' \wedge \varphi(\bar{\imath})$ for the input relation $\sim_{in}$ and $\bar{o} = \bar{o}'$ for the output relation $\sim_{out}$.

If a revision adds or removes existing variables from the declarations, the canonical relations to be considered are the conjoined equalities between all signals shared by both revisions (i.e., the variables in $\Delta_P \cap \Delta_Q$). This is called *projected equivalence*. The introductory example in Sect. 2 is a projected equivalence if the metallic detector is assumed absent in the first version and only introduced in the second. Another example of relational equivalence is shown in the case study in Sect. 6.

4 Environment Models to Increase Precision

False alarms can occur if the two revisions of a PLC program behave differently on inputs that cannot actually occur in the application. For example, the crane from Sect. 2 can never be in more than one of the positions A,B,C at the same time. Assuming correct working of the sensors, not more than one of the Boolean input variables A, B, C can be true at the same time. Thus, it would be irrelevant if the two program revisions were to react differently in case A and B were signaled simultaneously but would still be equivalent for all feasible inputs. It is therefore sensible to add such knowledge on the possible sensor inputs as assumptions to the process and perform a conditional regression verification. In the example, it is possible to encode the assumption in form of the LTL condition $\mathbf{G}(\neg(A \wedge B) \wedge \neg(B \wedge C) \wedge \neg(A \wedge C))$.

But in more involved cases, it is difficult or error-prone to express properties of the physical system correctly in form of conditions on the PLC inputs. Then it is better to use a model of the environment which uses output of PLC program as input. This restricts the search space, increases precision of regression verification and avoids false alarms. Fig. 2 depicts a model of the crane restricting the input space for the variables corresponding to the crane's position. Besides the three states for positions A, B, C in which only the corresponding PLC input variable is true, there are three intermediate states between the positions where none of variables is true. The crane behavior model shows that when the crane turns to

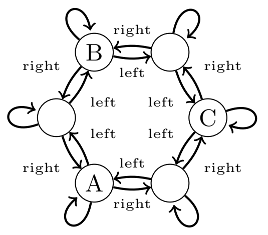

Fig. 2. Finite automaton modeling crane position sensor readings.

the right from position A to position B, first variable A is true, then no variable is true, and then B is true.

By making environment models non-deterministic (like in the example) one can abstract from details like concrete numbers of waiting cycles.

One evolution scenario of the case study in Sect. 7 describes a case where the PLC program revisions are only equivalent if an environment model is used.

5 Regression Verification Method and Toolchain

This section reports on how we achieve regression verification for PLC software by construction of a verification condition from two PLC program revisions, the equivalence relations $\sim_{in}, \sim_{out}$, the condition φ, and environment models.

The workflow of our method – shown in Fig. 3 – covers several transformation steps. The resulting verification condition consisting of a transition system and a property is presented to a model checker that can come back with three possible results: First, it may report that the verification property holds for the transition system in which case the two PLC programs are trace equivalent (modulo the condition, relations, and environment models). Second, it may report a counterexample with a concrete (finite) input trace that leads to the equivalence violation. There are no "false positives": Every reported violation uncovers a case of unequal behavior. However, it may be that the environment is not modeled precisely enough, and that the failure is a false alarm in the sense that it cannot occur in practice with the real hardware. The variables range over finite datatypes and the model checking problem is, in theory, decidable. Depending on the size and complexity of the verification condition, it is still possible that the model checker runs out of resources (time or memory) and does not come back with an answer, which is the third possible result.

5.1 From PLC Code to Model Checker Input

The IEC 61131-3 standard [19] defines two textual and three graphical PLC programming languages. According to the ARC industry advisory group [1], the use of PLC systems compliant with IEC 61131-3 currently is and will remain the state of industrial practice for the next five to ten years. We consider input PLC programs written in the textual language Structured Text (ST) or in the graphical language Sequential Function Chart (SFC). ST has seen the greatest increase in adoption [1]. It perhaps best embraces the growing complexity of PLC programming.

For a uniform treatment of programs regardless of the particular language, we define an intermediate language into which we translate all incoming programs.

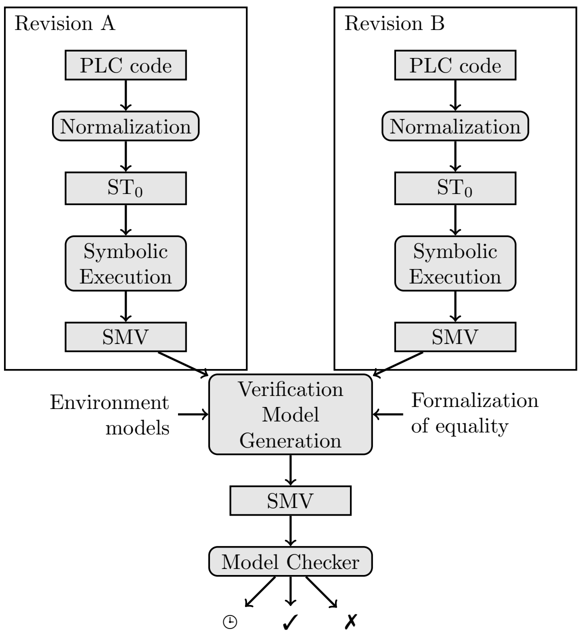

Fig. 3. Overview over the regression verification method.

This language is a sublanguage of ST called ST_0. Despite their notational differences, programs in all 61131-3 programming languages can be represented in ST_0 (provided they do not have unbounded loops). PLC programs are time-critical, and they are required to finish within their cycle time. It is therefore reasonable to assume that programs do not contain loops with an unbounded number of iterations.

The language ST_0 is essentially the loop- and call-free fragment of ST reduced to fewer, more basic datatypes. The only types of statements in ST_0 are assignments and if-then-else conditionals. During normalization to ST_0, loops are fully unwound and function block invocation are inlined. We require that for all loops a bound for the number of iterations can be statically computed from the code, so that this unwinding is always possible. Inlining is also feasible since recursion is not featured in the IEC 61131-3 framework.

To demonstrate our method, we implemented such translations to ST_0 for ST and SFC. The translation from SFC to ST_0 is problematic since the standard leaves many semantical issues unanswered. We resolved this issue by following the formal semantics for SFCs given in [4] when translating SFC to ST_0.

The normalized code in ST_0 is symbolically executed to derive a state transition system as model checker input. Naïve implementations of symbolic execution or other verification condition generation algorithms like weakest precondition calculus may produce program representations whose size is exponential in that of the original program. This is due to an explicit enumeration of all possible paths through the program. Since ST_0 programs that result from translating SFC code involve many consecutive and nested if-statements to encode the original state machine, the number of paths through the program is huge and explicitly enumerating them is infeasible. For example, the last scenario (Ev14) of our case study (Sect. 6) yields some 13 billion paths, such that the resulting proof obligation would not fit into the available memory.

Instead we produce a smaller program representation by not explicitly enumerating all paths but following the concept of Φ-nodes (known from static single assignment [10]) to merge the effects of the branches of an if-statement. This approach is also similar to the weakest-precondition-calculus optimization presented in [12].

During symbolic execution, a symbolic variable map $\mathcal{V} : \mathit{Vars}_\Delta \to \mathit{Terms}_\Delta$ is modified, which assigns to all declared variables their current symbolic value (a term). Stmt_Δ denotes the set of all ST_0-statements, and $t^\mathcal{V}$ is the symbolic evaluation of an expression t in the symbolic variable assignment $\mathcal{V}$.

Definition 5 (Symbolic Execution). *Symbolic execution of ST_0 code is the operator* $se : (\mathit{Vars}_\Delta \to \mathit{Terms}_\Delta) \times \mathit{Stmt}_\Delta \to (\mathit{Vars}_\Delta \to \mathit{Terms}_\Delta)$ *with*

$$\begin{aligned} se(\mathcal{V}, v := t) &:= \mathcal{V}[v := t^\mathcal{V}] \\ se(\mathcal{V}, S;T) &:= se(se(\mathcal{V}, S), T) \\ se(\mathcal{V}, \text{if } c \text{ then } S \text{ else } T) &:= \Phi(c^\mathcal{V}, se(\mathcal{V}, S), se(\mathcal{V}, T)) \end{aligned}$$

where the map $\Phi(c, \mathcal{V}_1, \mathcal{V}_2) : \mathit{Vars}_\Delta \to \mathit{Terms}_\Delta$ is, for all $v \in \mathit{Vars}_\Delta$, defined by:

$$\Phi(c, \mathcal{V}_1, \mathcal{V}_2)(v) := \begin{cases} \mathcal{V}_1(v) & \textit{if } \mathcal{V}_1(v) = \mathcal{V}_2(v) \\ \text{if } c \text{ then } \mathcal{V}_1(v) \text{ else } \mathcal{V}_2(v) & \textit{otherwise} \end{cases}$$

Essentially, this transformation moves the conditions of if-then-else statements into the variable assignment in form of if-then-else expressions. While this procedure cannot guarantee that the result is not exponentially larger than the input, our experiences show that the results are acceptable in practice.

The state transition system for a program P is computed as follows: The operator se is applied to the instructions Π of P with the identity mapping id_Δ as the starting point, resulting in the symbolic variable map $se(\Pi, id_\Delta)$. The symbolic assignments in this map provide the state transition definitions for the state variables and the output terms for the output variables.

5.2 Encoding Regression Verification

The proof obligation handed to the symbolic model checker consists of a state transition system and a property that is to be proved an invariant for it. The state transition system is a composition of the two systems that result from translating the two PLC program revisions P and Q and the models for the environment as introduced in Sect. 4.

All variables of the input spaces I_P and I_Q make up the input variables of the combined model. If Δ_P and Δ_Q share common input variables, these can also be shared in the combined model, thus reducing the input state space size for model checking.

If the sensor readings are constrained by an environment model, the input signals of that model are input signals of the entire state transition system while input signals of the PLC programs corresponding to sensor readings are taken from the outputs of the environment model. In the example environment model for the crane positions (Fig. 2), the PLC program takes the three inputs A, B, C from position sensors, while the composed verification model merely takes as input the indeterministic choice whether to remain in the current model state or whether to move on a step. This has two effects: (1) The input space size is reduced and (2) the modeling is more precise.

The condition φ from Def. 3 and the input and output relations $\sim_{in}, \sim_{out}$ from Def. 4 make up the invariant that is part of the model checking proof obligation. In the current version of our toolchain, we require that the condition φ can be expressed in LTL by a formula of the form $\mathbf{G}\,\psi$, where ψ is a propositional formula over the input variables in I_P without modal operators. That is, it must be possible to express the desired condition on the input sequence as a property of individual inputs. Correspondingly, we require that the relations $\sim_{in}, \sim_{out}$ can be expressed by LTL formulas $\sim_{in} = \mathbf{G}\,\tau_{in}$ resp. $\sim_{out} = \mathbf{G}\,\tau_{out}$, where τ_{in} and τ_{out} are propositional formulas over the variables in $I_P \cup I_Q$ resp. $O_P \cup O_Q$. We then employ a fresh internal state variable *pre* : *bool* to model the temporal condition within the invariant as follows:

$$\mathbf{init}(\mathit{pre}) := \mathit{true} \tag{1}$$

$$\mathbf{next}(\mathit{pre}) := \mathit{pre} \land \psi \land \tau_{in} \tag{2}$$

$$\mathbf{invariant} \quad \mathit{pre} \rightarrow \tau_{out} \tag{3}$$

The variable *pre* is initialized to true (1) and is invalidated (2) as soon as input values violate either the condition (ψ) or the input relation (τ_{in}). If the guarded invariant (3) holds for the transition system, then the equivalence of the two programs is guaranteed. What in fact is proved using the auxiliary variable *pre* is the LTL property $(\neg\psi \lor \neg\tau_{in})\ \mathbf{R}\ \tau_{out}$ stating that the output relation τ_{out} must hold at least as long as neither the condition ψ nor the relation τ_{in} have been violated ($\mathbf{R}$ is the "release" operator of LTL). This entails relational equivalence between P and Q.

All relations and conditions occurring in our case study fall into the restricted category of specifications described above. Although this is not implemented at the moment, other classes of LTL constraints can be used in our method by encoding them as invariants along the lines of [27].

5.3 Coupling Invariants

Modern model checkers allow the application of state abstraction methods (like IC3) to find proofs for safety properties more efficiently. Regression verification using symbolic model checkers with such abstractions is particularly promising, since the two software revisions are closely related if the newer one results from the adaptation of the older one to a new application scenario. In such cases, it is likely that the old and the new version of the program have a similar – yet not necessary equal – encoding of their state spaces.

The upcoming abstraction theorem allows us to reason about safety properties of two PLC programs P and Q using an invariant $\mathit{Inv} : S_P \times S_Q \rightarrow \mathit{bool}$ over their state spaces S_P and S_Q. Such a predicate, building a bridge between the state spaces, is called a *coupling predicate.*

Theorem 1 (Coupling Invariant Abstraction). *We consider two PLC programs P and Q with common input space I, common output space O, and state spaces S_P and S_Q. Let $s_0 \in S_P$ and $s'_0 \in S_Q$ be the initial states.*

Then, P and Q are (perfectly) trace equivalent if and only if there exists a coupling predicate $\mathit{Inv} : S_P \times S_Q \rightarrow \mathit{bool}$ such that, for all states $s \in S_P, s' \in S_Q$ and inputs $i \in I$,

1. *$Inv(s_0, s_0')$ holds,*
2. *$Inv(s, s')$ implies $Inv(t, t')$,*
3. *$Inv(s, s')$ implies $o = o'$,*

where $(t, o) = \rho(P)(s, i)$ and $(t', o') = \rho(Q)(s', i)$.

Similar theorems can be formulated for PLC programs that are relationally equivalent.

The more similar the state space encodings of the old and the new program version are, the closer the coupling predicate is to equality on the state spaces. This becomes evident when a PLC program P is verified against itself. In this case, the equality relation itself can be used as coupling predicate and satisfies the conditions in Theorem 1 regardless of what P computes.

Development of PLC programs is often an incremental process, i.e., the new revision results from a modification of the code in the old version. Often, parts of the state are not affected by the changes (and behave like in the old revision) whereas other parts are affected. An inductive invariant implying equivalence then comprises equality between the unmodified state variables, and a more general coupling invariant must be generated only for the affected variables.

The regression verification method using invariants is complete, but the user of the verification tool would have to find and formalize all coupling invariants which can be large and unintuitive. Instead, we rely upon the capabilities of state-of-the-art symbolic model checkers to automatically infer inductive invariants. In our case, the required system invariant (3) (which usually is not inductive itself) is used as a starting point for an interpolant-based search for a stronger inductive invariant that implies the one given in the problem specification.

We show in our case study that even with large state spaces, this state abstraction mechanism allows us to prove equivalence of non-trivial programs. The model checker nuXmv is capable of coming up with the required coupling predicates using Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) [7, 24]. If this invariant generation mechanism is switched off, the tool relies on more traditional symbolic model checking techniques. Then, even the smaller ones of the problems in the case study could not be solved.

In cases where the search for an inductive invariant takes too long, parts of the coupling invariant can be specified manually (within (3)) – the workload for the invariant generation can thus be shared between user and model checker.

6 Case Study

We have evaluated our approach by applying it to the benchmark evolution scenarios of the Pick-and-Place Unit (PPU), which is illustrated in Fig. 4. The PPU is an open case study for the machine manufacturing domain [35]. Despite being a bench-scale, academic demonstration case, the PPU is complex enough to demonstrate selected challenges that arise during engineering of automated production systems. To explore evolution in this context, sixteen scenarios (i.e., variants of the PPU) covering different aspects of evolution have been defined [22, 36]. There are both pure software changes as well as changes that incorporate adaptations to the mechanics and automation hardware of the PPU.

For all of the scenarios developed for the PPU, both the structure and the behavior of the PPU are documented using the Systems Modeling Language

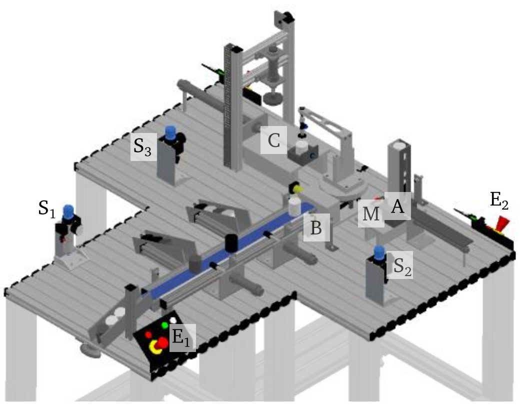

Fig. 4. Schematic of the hardware setup of the PPU case study [35].

(SysML) [35]. Also, IEC 61131-3 automation software code for the PLC is available for each evolution scenario – implemented in CODESYS2, an industrial development tool for automation software executable on PLCs. The PPU has 22 digital input, 13 digital output, and 3 analogue output signals and defines a number of simple discrete event automation tasks [33].

In the following, we discuss three evolution scenarios from the PPU and show how they can be subject to regression verification. More details can be found in [38]; see Table 1 for the time required for verification.

Conditional equivalence. The evolution scenario Ev3 in [35] has been used as the introductory example in Sect. 2 in a much simplified version. In the full scenario, the new stamping hardware for metallic products brings with it a new emergency stop button E_2 (triggering the same emergency logic as the existing button E_1) and a new start switch S_3 (complementing S_1 and S_2 already present). Only after *all* start switches have been pressed, the plant starts processing workpieces. Trace equivalence between the two revisions of this evolution step can only be shown for traces where these new components do not influence the flow of signals already present in the old software. This is the case if (1) no metallic workpiece is ever detected at M. (2) button E_2 is only pressed if simultaneously E_1 is also pressed, and (3) S_3 is not activated after the other switches S_1 and S_2 have been pressed. The LTL formula over the corresponding input variables for conditional equivalence of the PLC programs is

$$\mathbf{G}(\neg M \;\wedge\; (E_2 \rightarrow E_1) \;\wedge\; (S_1 \wedge S_2 \rightarrow S_3))$$

Using this condition, equivalence can indeed be proved by our toolchain.

Relational equivalence. In evolution step Ev14, the three position sensors at A, B and C are replaced by a single angle transmitter that continuously reports the angular position of the crane (in degrees). The observational behavior is to remain the same after the evolution. The input spaces for the PLC programs differ such that the equivalence that can be established must be relational.

In correspondence with the hardware setup (see Fig. 4) and the requirements of the production system, we model the relation that binds the old Boolean position inputs A, B, C to the new angular input α as

$$\mathbf{G}((A \leftrightarrow 0 \leq \alpha \leq 5) \wedge (B \leftrightarrow 90 \leq \alpha \leq 95) \wedge (C \leftrightarrow 180 \leq \alpha \leq 185)) \ .$$

In the thus defined input relation $\sim_{in}$ each position switch corresponds to a 5° interval in the angular input space. This also shows that relations in our approach can be more complex than just a biunique mapping between values.

Using an environment model. In evolution scenario Ev6, the hardware remains unmodified, but the software is changed to optimize the handling of non-metallic workpieces (see [35] for details). The PLC programs before and after the optimization should be equivalent for traces where only metallic or only non-metallic workpieces are detected, but the programs are *not* equivalent. An inspection of the code reveals that a condition within an SFC has been reformulated. As a first guess one could assume that the two conditions are equivalent and use this as condition for the conditional equivalence proof. Indeed, the equivalence proof succeeds using that assumption (Ev6+A for both cases, Ev6+$\mathrm{A_m}$ for metallic and Ev6+$\mathrm{A_{nm}}$ for non-metallic pieces only). However, using an ad-hoc assumption about the input state is not satisfactory even if it could be justified by a manual inspection. Instead, a more intuitive and convincing item, an environment model of the crane (essentially the one shown in Fig. 2) can be added, using which the PLC programs are proven equivalent with (Ev6+AEM) and even without the assumption (Ev6+EM).

Results. Using our method and toolchain, automatic regression verification was successful for all scenarios from the PPU case study.

Table 1 shows statistics for our experiments with the PPU. The evolution scenarios were verified using nuXmv version 1.0.1 on an Intel Dual-Core with 2.7 GHz and 4 GB RAM running OpenSUSE 12.2.

Not all evolution scenarios include a modification of the software. The scenarios for which the equivalence verification is trivial have been omitted from the table. The verification times for the same problem on the same machine may vary considerably in multiple runs due to random choices in the symbolic model checker which have a great impact on the verification time.

The regression verification method can not only be used for verifying equivalence of PLC programs up to intended differences, but unintentional differences between programs can also be found using our approach. The evaluation of our approach revealed a few unintentional regressions in the PPU. In four cases, new intermediate code blocks are added into SFCs that cause a regression by delaying the system answer one cycle for each workpiece. Since the cycle time is very short in the PPU (4 ms), the discrepancy between the programs was not found by testing. Moreover, regression verification discovered that a fix for a safety violation was not applied to an earlier version in the PPU evolution sequence. It is possible that the crane tries to grab a workpiece while it is still in motion which might under very unfortunate circumstances cause damages.

7 Related Work

The verification of PLC programs w.r.t. temporal logic specifications (for safety, liveness, and time properties) has been subject of a number of publications already. The paper [40] gives an overview of the field, and the survey [21] discusses transformation processes for program languages to verifiable models. Various translations from IEC 6113-3 languages into the input languages of model checkers have been presented: Brinksma et al. [8] present a translation of SFCs into

Table 1. Results of the experiments. **scenario** is the name of the evolution scenario in [35], **in** is the size of the sensor input space in bits, **state** the size of the state space in bits, **min/max** show the minimum and maximum time needed for verification using nuXmv in seconds (s), minutes (m) or hours (h). +EM indicates that an environment model has been used.

scenario	**in**	**state**	**min**	**max**	**scenario**	**in**	**state**	**min**	**max**
Ev1	10	140	4 s	8 s	Ev6+EM	11	299	2 m	21 m
Ev1+EM	12	146	7 s	12 s	Ev8	20	289	13.7 m	20.9 m
Ev2	11	141	4 s	8 s	Ev9	20	305	50.5 m	1.3 h
Ev3	19	246	9 s	17 s	Ev10	23	365	13 s	24 s
Ev6+A	19	284	15.1 m	155.4 h	Ev11	28	576	3.5 h	6.3 h
Ev6+A_m	19	284	8.9 m	9.1 h	Ev12	34	860	22.2 h	56.4 h
Ev6+A_{nm}	19	284	18.1 m	13 h	Ev13	34	1225	21.9 h	21.9 h
Ev6+AEM	11	299	25.7 m	104.1 h	Ev14	47	1663	22.1 h	22.1 h

Promela input for the SPIN model checker [17]; De Smet et al. [28] translate all languages within IEC 61131-3 into input for the symbolic model checker Cadence-SMV [25]; and Bauer et al. [3] translate SFCs into timed automata to be used with UPPAAL [5]. This model checker is also used to verify properties of continuous function charts (CFC) in [37]. In [4, 6] a unifying semantics for SFC is given where the ambiguities of the standard are addressed in a formal fashion.

Süflow and Drechsler [30] present a framework to verify that the *same* program behaves equivalently on *different* PLC platforms; a scenario closely related to ours. The authors employ a SAT solver to verify the arising proof conditions.

Strichman and Godlin [13–15, 29] coined the term *regression verification* and presented a verification methodology based on replacing function calls by uninterpreted function symbols within a bounded software model checking framework for C programs. In [13] they define “reactive equivalence,” which is closely related to our notion of perfect trace equivalence. In earlier work [11], we presented an automated approach to regression verification based on invariant generation using Horn clauses. Many other approaches [2, 16, 31, 32, 39] exist to regression verification for imperative programming languages.

Equivalence checking is an established issue for the verification of hardware circuits. In *sequential equivalence checking* the perfect trace equivalence between clocked circuits is analysed; see [18] or [20] for an overview. Lu and Cheng [23] present an approach based on inferred invariants, conditional or relational equivalence are not considered.

8 Conclusion and Future Work

We have presented a method and toolchain for the automatic regression verification of PLC software by means of a symbolic model checker. In this process, the old software revision serves as specification for the new one. *Conditions* can be specified under which systems must behave equivalently, *relations* can be specified how the equivalence is to be understood, and *models of environment* can be added to make the process more precise.

Evaluation proved our method to be applicable to non-trivial PPC software. Automatic regression verification was successful for all scenarios from the PPU case study. The evaluation also showed that the use of Φ-nodes in the translation from PPU code to model checking input as well as the automatic generation of coupling invariants is indispensable for non-trivial programs.

Currently, our toolchain supports notions that compare PLC behavior cycle by cycle. Future work will allow for conditions and relations to relate variables of different cycles. Another interesting path of investigation is the use of abstractions to factor out parts of PLCs that have not been touched by evolution and need not be proved equivalent.

Acknowledgement. The authors thank Alberto Griggio for his valuable input on the effective use of nuXmv and Vladimir Klebanov for his feedback on an earlier version of this paper.

This work was supported by the DFG (German Research Foundation) in Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

1. ARC Advisory Group. PLC & PLC-based PAC worldwide outlook: Five year market analysis and technology forecast through 2016, 2011.
2. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *Proc., Int. Symp. on Formal Methods (FM)*, LNCS. Springer, 2011.
3. N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg. Verification of PLC programs given as sequential function charts. In *Integration of Software Specification Techniques for Applications in Engineering*, LNCS 3147, pages 517–540. Springer, 2004.
4. N. Bauer, R. Huuck, B. Lukoschus, and S. Engell. A unifying semantics for sequential function charts. In *Integration of Software Specification Techniques for Applications in Engineering*, LNCS 3147, pages 400–418. Springer, 2004.
5. G. Behrmann, K. Larsen, O. Moller, A. David, P. Pettersson, and W. Yi. Uppaal: Present and future. In *Decision and Control, 2001. Proc. of the IEEE Conf.*, 2001.
6. S. Bornot, R. Huuck, and B. Lukoschus. Verification of sequential function charts using SMV. In H. R. Arabnia, editor, *PDPTA*. CSREA Press, 2000.
7. A. R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, LNCS 6538, pages 70–87. Springer, 2011.
8. E. Brinksma, A. Mader, and A. Fehnker. Verification and Optimization of a PLC Control Schedule. *Software Tools for Technology Transfer*, 4(1):21–33, 2002.
9. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In *CAV*, LNCS 8559. Springer, 2014.
10. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proc. ACM Symp. on Principles of Programming Languages*, POPL '89. ACM, 1989.
11. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14. ACM, 2014.
12. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Principles of Programming Languages*. ACM, 2001.
13. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.
14. B. Godlin and O. Strichman. Regression verification. In *Proc., 46th Annual Design Automation Conference*. ACM, 2009.
15. B. Godlin and O. Strichman. Regression verification: Proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.
16. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Towards modularly comparing programs using automated theorem provers. In *Proc., Int. Conf. on Automated Deduction*, LNCS 7898. Springer, 2013.

17. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.
18. S.-Y. Huang and K.-T. Cheng. *Formal Equivalence Checking and Design DeBugging*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.
19. International Electrotechnical Commission. IEC 61131-3: Programmable Logic Controllers – Part 3: Programming Languages, 2009.
20. A. Kuehlmann and C. van Eijk. Combinational and sequential equivalence checking. In *Logic Synthesis and Verification*. Springer, 2002.
21. S. Lampérière-Couffin, O. Rossi, J.-M. Roussel, and J.-J. Lesage. Formal validation of PLC programs: a survey. In *European Control Conference*, 1999.
22. C. Legat, J. Folmer, and B. Vogel-Heuser. Evolution in industrial plant automation: A case study. In *Industrial Electronics Society, IECON*. IEEE, 2013.
23. F. Lu and K.-T. Cheng. SEChecker: A sequential equivalence checking framework based on k-th invariants. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(6):733–746, 2009.
24. K. McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification*, LNCS 2725. Springer, 2003.
25. K. L. McMillan. *Symbolic model checking*. Kluwer, 1993.
26. A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.
27. V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *STTT*, 5(2-3):185–204, 2004.
28. O. D. Smet, S. Couffin, O. Rossi, G. Canet, J.-J. Lesage, P. Schnoebelen, and H. Papini. Safe programming of PLC using formal verification methods. In *4th Int. PLCopen conference on Industrial Control Programming*, 2000.
29. O. Strichman. Regression verification: Proving the equivalence of similar programs. In *Computer Aided Verification*, LNCS 5643. Springer, 2009.
30. A. Süflow and R. Drechsler. Verification of PLC programs using formal proof techniques. In *FORMS/FORMAT*, 2008.
31. S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3):11:1–11:35, 2012.
32. S. Verdoolaege, M. Palkovic, M. Bruynooghe, G. Janssens, and F. Catthoor. Experience with widening based equivalence checking in realistic multimedia systems. *J. Electronic Testing*, 26(2):279–292, 2010.
33. B. Vogel-Heuser. Usability experiments to evaluate uml/sysml-based model driven software engineering notations for logic control in manufacturing automation. *Journal of Software Engineering and Applications*, 7(11):943–973, 2014.
34. B. Vogel-Heuser, C. Diedrich, A. Fay, S. Jeschke, S. Kowalewski, M. Wollschlaeger, and P. Göhner. Challenges for software engineering in automation. *Journal of Software Engineering and Applications*, 7(5), May 2014.
35. B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit. Technical Report TUM-AIS-TR-01-14-02, TUM, 2014.
36. B. Vogel-Heuser, C. Legat, J. Folmer, and S. Rösch. Challenges of parallel evolution in production automation focusing on requirements specification and fault handling. *Automatisierungstechnik*, pages 758–770, 2014.
37. A. Wardana, J. Folmer, and B. Vogel-Heuser. Automatic program verification of continuous function chart based on model checking. In *Industrial Electronics, 2009. IECON '09. 35th Annual Conference of IEEE*, pages 2422–2427, Nov 2009.
38. A. Weigl. Regression verification of programmable logic controller software. Master's thesis, Karlsruhe Institut of Technology, January 2015.
39. Y. Welsch and A. Poetzsch-Heffter. Verifying backwards compatibility of object-oriented libraries using Boogie. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 35–41. ACM, 2012.
40. M. B. Younis and G. Frey. Formalization of existing PLC programs: A survey. In *Proceedings of CESA*, 2003.